HPC: HW1 Report

2017-19841 최창민

1. Theoretical Peak Performance of CPU

```
shpc145@login0:~$ srun -p shpc -N 1 lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              64
On-line CPU(s) list: 0-63
Thread(s) per core:  2
Core(s) per socket:  16
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               85
Model name:          Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz
Stepping:            7
CPU MHz:             1446.550
CPU max MHz:         2101.0000
CPU min MHz:         800.0000
BogoMIPS:            4200.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            22528K
NUMA node0 CPU(s):   0-15,32-47
NUMA node1 CPU(s):   16-31,48-63
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mt
 tm pbe syscall nx pdpelgb rdtscp lm constant_tsc art arch_perfmon
pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16
_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fa
 stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid fsgsbase
 avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd
p_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts pku ospke
```

```
COLUMNS
    Note that topology elements (core, socket, etc.) use a sequential unique ID
    starting from zero, but CPU logical numbers follow the kernel  where  there
    is no guarantee of sequential numbering.

    CPU    The logical CPU number of a CPU as used by the Linux kernel.

    CORE   The logical core number.  A core can contain several CPUs.

    SOCKET The logical socket number.  A socket can contain several cores.

    BOOK   The logical book number.  A book can contain several sockets.

    DRAWER The logical drawer number.  A drawer can contain several books.

    NODE   The logical NUMA node number.  A node can contain several drawers.

    CACHE  Information about how caches are shared between CPUs.

    ADDRESS
           The physical address of a CPU.

    ONLINE Indicator  that shows whether the Linux instance currently makes use
           of the CPU.

    CONFIGURED
           Indicator that shows if the hypervisor has allocated the CPU to  the
           virtual  hardware  on  which the Linux instance runs.  CPUs that are
           configured can be set online by the  Linux  instance.   This  column
           contains  data  only  if your hardware system and hypervisor support
           dynamic CPU resource allocation.

    POLARIZATION
           This column contains data for Linux instances that  run  on  virtual
           hardware  with a hypervisor that can switch the CPU dispatching mode
           (polarization).  The polarization can be:

           horizontal  The workload is spread across all available CPUs.

           vertical    The workload is concentrated on few CPUs.

           For vertical polarization, the column also shows the degree of  con-
           centration, high, medium, or low.  This column contains data only if
           your hardware system and hypervisor support CPU polarization.

    MAXMHZ Maximum megahertz value for the CPU. Useful when lscpu  is  used  as
           hardware  inventory  information  gathering  tool.   Notice that the
           megahertz value is dynamic, and driven by CPU governor depending  on
           current resource need.

    MINMHZ Minimum megahertz value for the CPU.
```

a. Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz
b. 2 socket
c. 베이스클럭: 2.10 GHz / 부스트 클럭: 3.20 GHz
   항상 Boost clock frequency로 동작하기에는 발열/전력량, 전력효율/손상 등의 이슈가 있을 수 있기에 작업 부하가 강할 때만 boost clock frequency로 클럭을 높여서 동작함.
d. 물리코어: 16 / 논리코어: 32 // 물리코어를 사용해야하는데 이는 논리코어는 IO blocking같은 이슈가 있을 때 병렬적으로 더 작업을 돌릴 수 있다는 뜻이고 실제로는 1코어당 1개의 작업만 할 수 있기 때문
e. Instruction 하나 당 16개의 연산 X 클럭당 4개 연산 = 64개의 연산
f. CPU당 코어(16) X CPU 갯수(2) X Boost Clock Frequency (2.10GHz) X AVX512의 연산갯수(64) = 4300.8 GFLOPS

2. OpenMP Matrix Multiplication

병렬화 방식 코드 설명

```
void matmul(const float *A, const float *B, float *C, int M, int N, int K,
            int num_threads) {
 // TODO: FILL_IN_HERE
 // A: M x K
 // B: K x N
 // C: M x N
 // printf("HI\n");
 omp_set_num_threads(num_threads);
 float a;
 float sh[N * num_threads];
 #pragma omp parallel private(a)
 {
   int tid = omp_get_thread_num();
   int pos = tid * N;
   #pragma omp for nowait schedule(auto)
   for (int m = 0; m < M; ++m) {
     memset(&sh[pos], 0, N * sizeof(float));
     for (int k = 0; k < K; ++k) {
       a = A[k + K * m];
       for (int n = 0; n < N; ++n) {
         sh[n + pos] += a * B[n + N * k];
       }
     }
     memcpy(&C[N*m], &sh[pos], N * sizeof(float));
   }
 }
}
```

기본적으로 3중루프를 구성하여 해결하였습니다. 3중루프의 중첩 순서는, 제일 바깥쪽에 M을 순회, 중간에 K를 순회, 가장 안쪽에 N을 순회하였습니다. 이는 cache 의 spatial locality를 살리기 위해서 였습니다.

C[n + N * m] += A[k + K * m] + B[n + N * k] 에서 n 은 spatial locality를 살리기 가장 좋았고, k 는 A의 입장에서는 살리기 좋았으나 B의 입장에서는 비교적 불리했고 m은 가장 살리기 힘들었습니다. 때문에 M 순회의 안에 K 순회, K 순회 안에 N 순회를 넣었습니다.

그리고 반복적으로 사용되는 A[k + K * m]을 변수로 저장하여 N 순회를 하는 중 필요없는 A의 spatial locality를 사용하지 않는 것을 의도하였고, sh라는 shared array를 두어 C에 직접 적지 않고, 비교적 캐시에 올라가기 쉽도록 sh에 적도록 한 후 memcpy를 통해 빠르게 적는 것을 의도하였습니다. 또한 스레드간에 서로 sh를 침범하지 않도록 tid를 가져와서 position을 계산하여 sh를 접근하였습니다.

최종적으로 가장 바깥쪽 M순회에 omp for를 nowait, auto scheduling과 함께 걸어 병렬화하도록 했습니다.

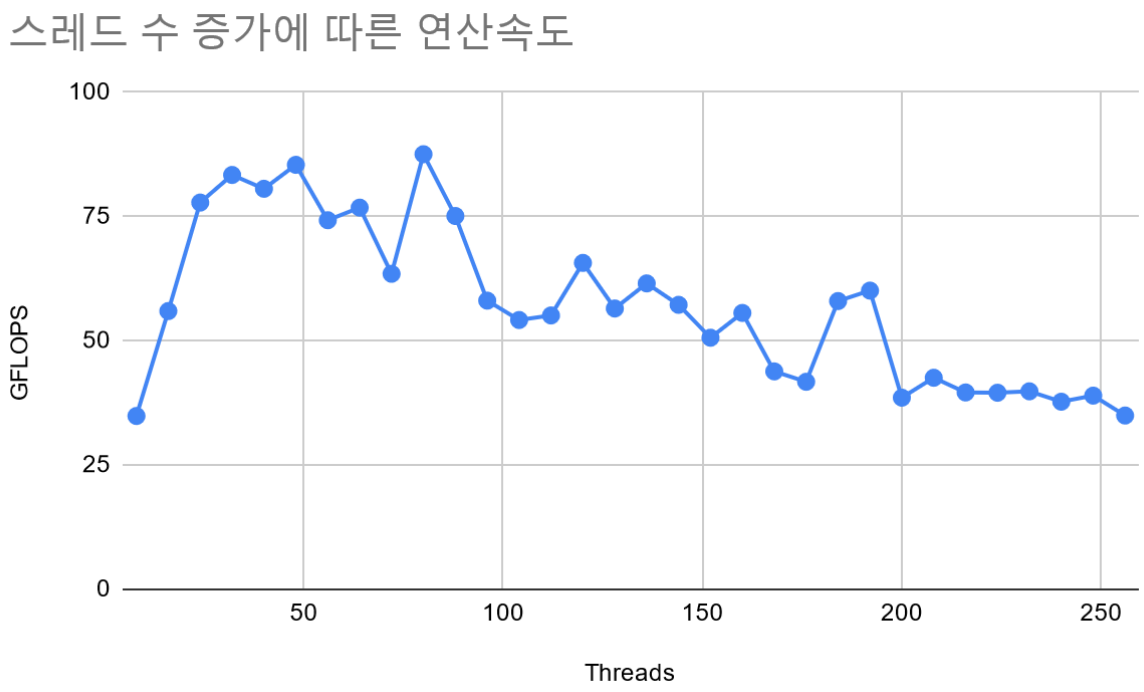
OpenMP에서 스레드의 생성

OpenMP에서 스레드의 생성은 Pragma 라는 Special Preprocessor Instruction을 써서 작성합니다. 이를 OpenMP를 지원하는 컴파일러와 함께 쓰면 기본적인 C 언어 스펙에 없는 동작들을 할 수 있고, 만약 지원하지 않는 컴파일러가 본다면 없는 코드로 생각하고 동작합니다.

OpenMP를 지원하는 컴파일러는 유저를 위해서 thread program과 synchronization을 작성해줍니다. 다만 자동으로 병렬화는 진행하지 않습니다.

OpenMP에서 스레드는 OpenMP 런타임 시스템에 의해서 관리되어 유저가 비교적 쉽게 사용할 수 있도록 돕습니다.

스레드 수 증가에 따른 행렬곱 성능



스래드를 8부터 256까지 8 간격으로 조절하여 성능을 측정하였습니다. 최대한 성능의 편차를 줄이기 위해서 a11노드만을 사용했으며, 반복횟수는 기본 값인 10회로 하여 측정하였습니다.

32스레드까지 증가하는 동안 성능이 증가했으며 32에서 48 스레드에서 어느정도 비슷한 성능을 보이다가 이후로는 점차 성능이 감소했습니다. 즉, 32스레드까지는 일정한추세로 성능이 증가했지만, 이후 48스레드까지는 비슷한 성능, 48스레드 이후로는 성능이 감소했습니다.

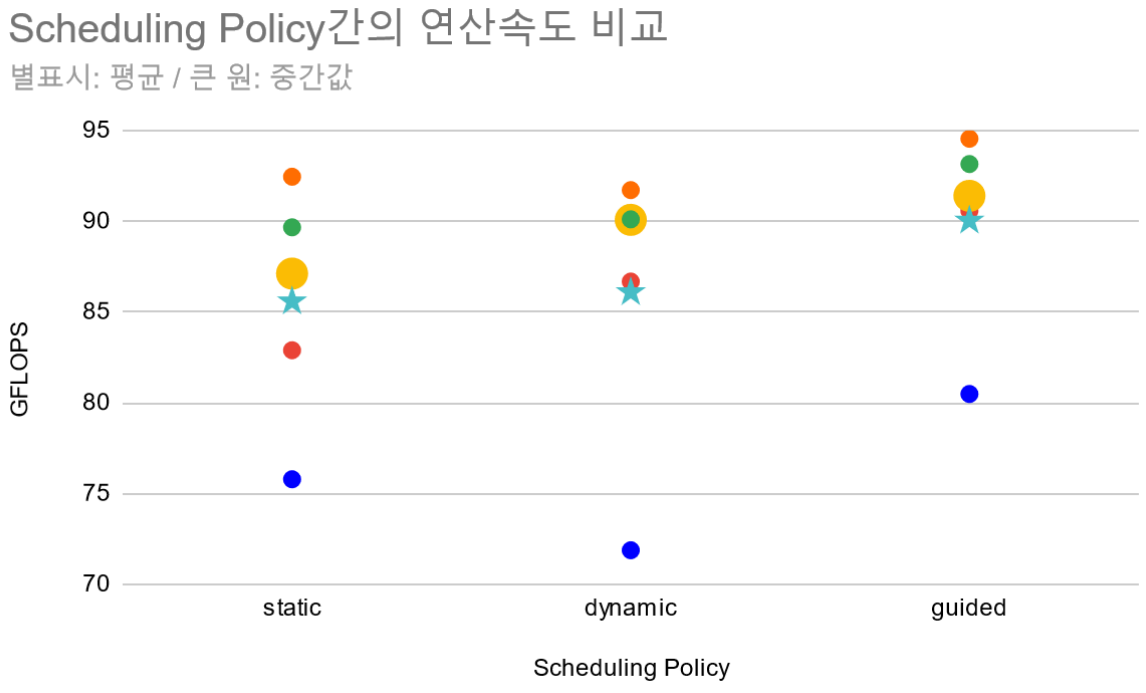
이는 어느정도까지는 스레드가 증가함에 따라 병렬화로 얻는 성능 gain이 멀티스레딩에서의 스케줄링, 동기화같은 overhead보다 더 컸지만 이후로는 overhead가 지나치게 커져서 overhead가 병렬화로 얻는 성능 gain을 압도하여 오히려 성능이 감소하는 것 때문입니다.

현재 성능과 성능의 이론치와 비교

가장 높은 성능을 보이는 스레드에서 행렬곱 성능은 대략 87.6 GFLOPS입니다. 이는 이론치인 4300.8 GFLOPS에 비하면 대략 2퍼센트 정도 되는 성능입니다. 이론치에 좀 더 가깝게 가기 위해서는, AVX512 연산이 매우 효율적인 만큼 이를 좀 더 활용해야 할 것으로 보입니다.

OpenMP Loop Scheduling

- static
    - 청크들을 static한 chunck size로 자른 다음 team안에 있는 스레드들에게 round-robin규칙으로 나누어 주는 것. 작업들이 균등할 것으로 예상될 때 주로 사용
    - 오버헤드가 적다는 장점이 있다.
- dynamic
    - 스레드가 작업을 다 한 후 다음 작업을 요청할 때 다음 청크를 주는 것.
    - 로드밸런싱에 있어서 장점이 있다.
- guided
    - static과 dynamic을 장점을 섞으려 한 것으로 큰 청크들로 나눴다가 스레드가 작업을 다 하고 돌아오면 좀 더 작은 청크를 작업용으로 주는 것.
    - static보다 로드밸런싱을 잘 하고 dynamic보다 오버헤드가 적다



실험은 제 코드에서 스케줄링 규칙만을 바꾸어서 진행하였습니다. 10번 반복하여 GFLOPS를 측정하는 것을 스케줄링 규칙당 5번씩 진행하고 이를 Plot해보았습니다. dynamic과 static의 평균값은 비슷했으나

중앙값은 dynamic이 더 높았습니다. guided는 dynamic과 중앙값은 비슷했으나 평균은 guided가 더 높았습니다. 종합적으로 봤을 때, 제 코드에서는 guided가 가장 효과적이었습니다.